The International Journal Of Engineering And Science (IJES)
||Volume|| 2 ||Issue|| 10 ||Pages|| 32-38 ||2013||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Masking the Data in Horizontal Aggregations in SQL to Prepare Data Sets for Data Mining Analysis

[1] V. Prashanthi, [2] K. Vinay Kumar
[1, 2] *Department of Computer Science, KITS Warangal*

----------ABSTRACT----------

Getting ready a facts set intended for analysis is normally the most difficult task in a very data exploration project, requesting many complex SQL queries, joining dining tables and aggregating columns. Existing SQL aggregations possess limitations to prepare data sets since they return just one column for each aggregated collection. In general, a major manual effort must build facts sets, when a horizontal layout is essential. We recommend simple, however powerful, techniques to generate SQL code to return aggregated columns in a very horizontal tabular page layout, returning a few numbers as opposed to one variety per short period. This new class involving functions is named horizontal aggregations. Horizontal aggregations assemble data sets using a horizontal denormalized page layout e. g. point-dimension, observation-variable, instance-feature, which is the normal layout needed by most data exploration algorithms. We recommend three fundamental techniques to evaluate horizontal aggregations: Exploiting the actual programming case construct; SPJ: Depending on standard relational algebra operators; PIVOT: Using the PIVOT driver, which emerges by several DBMSs. Experiments with large tables evaluate the planned query analysis methods.

Keywords— *Data Transformation, Pivoting, SQL.*

Date of Submission: 11th, October, 2013  Date of Acceptance: 30th, October, 2013

I. INTRODUCTION

In a relational database, especially with normalized tables, a significant effort is required to prepare a summary data set that can be used as input for a data mining or statistical algorithm. Most algorithms require as input a data set with a horizontal layout, with several records and one variable or dimension per column. That is the case with models like clustering, classification, regression and PCA; consult. Each research discipline uses different terminology to describe the data set. In data mining the common terms are point-dimension. Statistics literature generally uses observation-variable. Machine learning research uses instance-feature. This article introduces a new class of aggregate functions that can be used to build data sets in a horizontal layout denormalized with aggregations, automating SQL query writing and extending SQL capabilities. We show evaluating horizontal aggregations is a challenging and interesting problem and we introduce alternative methods and optimizations for their efficient evaluation.

As mentioned above, building a suitable data set for data mining purposes is a time-consuming task. This task generally requires writing long SQL statements or customizing SQL code if it is automatically generated by some tool. There are two main ingredients in such SQL code: joins and aggregations; we focus on the second one. The most widely-known aggregation is the sum of a column over groups of rows. Some other aggregations return the average, maximum, minimum or row count over groups of rows. There exist many aggregation functions and operators in SQL. Unfortunately, all these aggregations have limitations to build data sets for data mining purposes. The main reason is that, in general, data sets that are stored in a relational database or a data warehouse come from On-Line Transaction Processing (OLTP) systems where database schemas are highly normalized. But data mining, statistical or machine learning algorithms generally require aggregated data in summarized form. Based on current available functions and clauses in SQL, a significant effort is required to compute aggregations when they are desired in a cross tabular horizontal form, suitable to be used by a data mining algorithm.

Such effort is due to the amount and complexity of SQL code that needs to be written, optimized and tested. There are further practical reasons to return aggregation results in a horizontal cross-tabular layout. Standard aggregations are hard to interpret when there are many result rows, especially when grouping attributes have high cardinalities. To perform analysis of exported tables into spreadsheets it may be more convenient to have aggregations on the same group in one row e.g. to produce graphs or to compare data sets with repetitive information. OLAP tools generate SQL code to transpose results sometimes called PIVOT. Transposition can be more efficient if there are mechanisms combining aggregation and transposition together.

II. RELATED WORK

Pivot and Unpivot are complementary data manipulation operators that modify the role of rows and columns in a relational table. Pivot transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to determine the new column for a row, and another source column is used as the data for that new column. Unpivot provides the inverse operation, removing a number of columns and creating additional rows that capture the column names and values from the wide form. The wide form can be considered as a matrix of column values, while the narrow form is a natural encoding of a sparse matrix. Figure 1 demonstrates how Pivot and Unpivot can transform data between narrow and wide tables. For certain classes of data, these operators provide powerful capabilities to RDBMS users to structure, manipulate, and report data in useful ways.

Implementations of pivoting functionality already exist for the purpose of data presentation, but these operations are usually performed either outside the RDBMS or as a simple post-processing operation outside of query processing. Microsoft Excel, for example, supports pivoting. Users can perform a traditional SQL query against a data source, import the result into Microsoft Excel, and then perform pivoting operations on the results returned from that data source. Microsoft Access (which uses the Microsoft Jet Database Engine) also provides pivoting functionality. This pivot implementation is a post-processing operation through cursors. While existing implementations are certainly useful, they fail to consider Pivot or Unpivot as first-class RDBMS operations, which is the topic of this paper.

Inclusion of Pivot and Unpivot inside the RDBMS enables interesting and useful possibilities for data modeling. Existing modeling techniques must decide both the relationships between tables and the attributes within those tables to persist. The requirement that columns be strongly defined contrasts with the nature of rows, which can be added and removed easily. Pivot and Unpivot, which exchange the role of rows and columns, allow the *a priori* requirement for pre-defined columns to be relaxed. These operators provide a technique to allow rows to become columns dynamically at the time of query compilation and execution. When the set of columns cannot be determined in advance, one common table design scenario employs "property tables", where a table containing (id, propertyname, propertyvalue) is used to store a series of values in rows that would be desirable to represent columns.

Users typically use this design to avoid RDBMS implementation restrictions (such as an upper limit for the number of columns in a table or storage overhead associated with many empty columns in a row) or to avoid changing the schema when a new property needs to be added. This design choice has implications on how tables in this form can be used and how well they perform in queries. Property table queries are more difficult to write and maintain, and the complexity of the operation may result in less optimal query execution plans. In general, applications written to handle data stored in property tables cannot easily process data in the widse (pivoted) format. Pivot and Unpivot enable property tables to look like regular tables (and vice versa) to a data modeling tool. These operations provide the framework to enable useful extensions to data modelling.

Including Pivot and Unpivot explicitly in the query language provides excellent opportunities for query optimization. Properly defined, these operations can be used in arbitrary combinations with existing operations such as filters, joins, and grouping. For example, since Unpivot transposes columns into rows, it is possible to convert a filter (an operation that restricts rows) over unpivot into a projection (an operation that restricts columns) beneath it. Algebraic equivalences between Pivot/Unpivot and existing operators enable consideration of many execution strategies through reordering, with the standard opportunity to improve query performance.

Fig 1: Pivot and UnPivot

Item Table

Item Key	Item name
1	2001
2	2002
3	2003

Property Table

Item Key	Prop Name	Prop Value
1	Jan	100
1	Feb	110
1	Mar	120
2	Jan	150
2	Feb	200
2	Mar	250

Fig 2: Sample SQL Table

Furthermore, new optimization techniques can also be introduced that take advantage of unique properties of these new operators. Consideration of these issues provides powerful techniques for improving existing user scenarios currently performed outside the confines of a query optimizer.

We argue that pivoting operations can be performed more quickly and powerfully inside a RDBMS. By implementing these operations as relational algebra operators within a cost-based optimization framework, superior execution strategies can be considered. This design choice also allows other relational operations to be performed on the results of pivot and unpivot. Considerations of the interactions between pivot/unpivot and other operators yield more efficient orderings of operations over post-processing. The inclusion of these operations within the declarative framework of a SQL statement also allows consideration of additional access paths, such as indexes or materialized views, to more efficiently compute results. Consideration of Pivot and Unpivot within a cost-based optimizer framework provides opportunities for superior performance over existing approaches [1].

Although materialized view maintenance has been well-studied in the research literature, with rare exceptions, to date that published literature has ignored concurrency control. In fact, if we use generic concurrency control mechanisms, immediate materialized aggregate join view maintenance becomes extremely problematic ¾ the addition of a materialized aggregate join view can introduce many lock conflicts and/or deadlocks that did not arise in the absence of this materialized view. As an example of this effect, consider a scenario in which there are two base relations: the *lineitem* relation, and the *partsupp* relation, with the schemas *lineitem* (*orderkey*, *partkey*), and *partsupp* (*partkey*, *suppkey*). Suppose that in transaction T_1 some customer buys items p_{11} and p_{12} in order o_1, which will cause the tuples (o_1, p_{11}) and (o_1, p_{12}) to be inserted into the *lineitem* relation. Also suppose that concurrently in transaction T_2 another customer buys items p_{21} and p_{22} in order o_2. This will cause the tuples (o_2, p_{21}) and (o_2, p_{22}) to be inserted into the *lineitem* relation. Suppose that parts p_{11} and p_{21} come from supplier s_1, while parts p_{12} and p_{22} come from supplier s_2. Then there are no lock conflicts nor is there any potential for deadlock between T_1 and T_2, since the tuples inserted by them are distinct.

Suppose now that we create a materialized aggregate join view *suppcount* to provide quick access to the number of parts ordered from each supplier, defined as follows:

Create aggregate join view suppcount as select p.suppkey, count (*)
from lineitem l, partsupp p where l.partkey=p.partkey group by p.suppkey;

Now both transactions T_1 and T_2 must update the materialized view *suppcount*. Since both T_1 and T_2 update the same pair of tuples in *suppcount* (the tuples for suppliers s_1 and s_2), there are now potential lock conflicts [2].

This article studies aggregations involving percentages using the SQL language. SQL has been growing over the years to become a fairly comprehensive and complex database language. Nowadays SQL is the standard language used in relational databases. Percentages are essential to analyze data.

Percentages help understanding statistical information at a basic level. Percentages are used to compare quantities in a common scale. Even further, in some applications percent- ages are used as an intermediate step for more complex analysis. Unfortunately traditional SQL aggregate functions are cumbersome and inefficient to compute percentages given the amount of SQL code that needs to be written and the inability of the query optimizer to efficiently evaluate such aggregations. Therefore, we propose two simple percentage aggregate functions and important recommendations to efficiently evaluate them. This article can be used as a guide to generate SQL code or as a proposal to extend SQL with new aggregations.

Our proposed aggregations are intended to be used in On- Line Analytical Processing (OLAP) and Data Mining environments. Literature on computing aggregate functions is extensive. An important extension is the CUBE operator proposed. There has been a lot of research following that direction. Optimizing view selection for data warehouses and indexing for efficient access in OLAP applications are important related problems.

Let *F* be a relation having a primary key represented by a row identifier (RID), *d* categorical attributes and one numerical attribute: $F(RID;D_1; : : : ;D_d;A)$. Relation *F* is represented in SQL as a table having a primary key, *d* categorical columns and one numerical column. We will manipulate *F* as a cube with d dimensions and one measure.

Categorical attributes (dimensions) are used to group rows to aggregate the numerical attribute (measure). Attribute. A represents some mathematical expression involving measures. In general *F* can be a temporary table resulting from some query or a view.

This section introduces two SQL aggregate functions to compute percentages in a multidimensional fashion. The first aggregation is called vertical percentage and the second one is called horizontal percentage. The vertical percentage aggregation computes one percentage per row like standard SQL aggregations, and the horizontal percentage aggregation returns each set of percentages adding 100% as one row. Queries using percentage aggregations are called percentage queries. We discuss issues about percentage queries and potential solutions. We study the problem of optimizing percentage queries [3].

In this work we focus on data preprocessing, an important data mining topic that has received scant attention in the literature. Nowadays transaction processing and data warehousing of large databases are performed mostly by relational database systems (DBMSs). Analyzing large databases with data mining techniques is a different story. Despite the existence of many data mining techniques offered by a database system, there exist many data mining tasks that are routinely performed with external tools.

This is due, among other reasons, to the existence of advanced data mining programs (i.e. open-source, third party commercial software), the availability of mathematical libraries (e.g. LAPACK, BLAS), the lack of expertise of data mining practitioners to write correct and efficient SQL queries and legacy code. Therefore, the database system just acts as a data provider, and users write SQL queries to extract summarized data with joins and aggregations. Once raw data sets are exported they are further cleaned and transformed depending on the task at hand, outside the database system. Finally, when the data set has the desired variables (features) and an appropriate number of records (observations, instances), data mining models are iteratively computed, interpreted and tuned. From all the tasks listed above preparing, cleaning and transforming data for analysis is generally the most time-consuming task because it requires significant effort to convert normalized tables in the database into denormalized tabular data sets, appropriate for data mining algorithms. Unfortunately, manipulating data sets outside the database system creates many data management issues: data sets must be recreated and re-exported every time required tables change, security is compromised (an essential aspect today due to the Internet). Data sets need to be re-exported when new variables (features) are created, models need to be deployed inside the database system, and different users may have inconsistent version of the same data set in their own computers. Therefore, we defend the idea of transforming data sets and computing models inside a modern database system, exploiting the extensive features of the SQL language and enjoying the extensive data management capabilities provided by the database system. Our motivation to migrate data mining preprocessing into the database system, yields the following benefits. The database system provides powerful querying capabilities through SQL and 3rd party tools. Even further, the database system provides the best data management functionality (maintaining referential integrity, transaction processing, fault-tolerance, security). Additionally, the database system server is generally a fast computer and thus results can be obtained sooner and with less programming effort.

We present important practical issues to preprocess and transform tables in a database in order to build a data set suitable for analysis. These issues have been collected from several projects helping users migrate analysis performed in external data mining tools into a database system.

We present issues in three groups: creating and transforming variables for analysis; selecting (filtering) records for analysis; developing and optimizing SQL queries.

In general, the main objection from users against using SQL is to translate existing legacy code. Commonly such code has existed for a long time (legacy programs), it is extensive (there exist many programs) and it has been thoroughly debugged and tuned. Therefore, users are reluctant to rewrite it in a different language, given associated risks. A second complaint is that, in general, the database system provides good, but still limited, data mining functionality, compared to sophisticated data mining and statistical packages

III. SYSTEM MODEL

A. SPJ Method

The SPJ method is interesting from a theoretical point of view because it is based on relational operators only. The basic idea is to create one table with a vertical aggregation for each result column, and then join all those tables to produce F_H. We aggregate from F into d projected tables with d Select- Project-Join-Aggregation queries (selection, projection, join, aggregation). Each table F_I corresponds to one subgrouping combination and has $\{L_1, \ldots, L_j\}$ as primary key and an aggregation on A as the only non-key column. It is necessary to introduce an additional table F_0, that will be outer joined with projected tables to get a complete result set. We propose two basic sub-strategies to compute F_H. The first one directly aggregates from F. The second one computes the equivalent vertical aggregation in a temporary table F_V grouping by $L_1, \ldots, L_j, R_1, \ldots, R_k$. Then horizontal aggregations can be instead computed from FV , which is a compressed version of F, since standard aggregations are distributive [9]. We now introduce the indirect aggregation based on the intermediate table F_V , that will be used for both the SPJ and the CASE method. Let F_V be a table containing the vertical aggregation, based on $L_1, \ldots, L_j, R_1, \ldots, R_k$. Let $V()$ represent the corresponding vertical aggregation for $H()$.

B. Case Method

For this method we use the ”case” programming construct available in SQL. The case statement returns a value selected from a set of values based on boolean expressions. From a relational database theory point of view this is equivalent to doing a simple projection/aggregation query where each nonkey value is given by a function that returns a number based on some conjunction of conditions. We propose two basic sub-strategies to compute F_H. In a similar manner to SPJ, the first one directly aggregates from F and the second one computes the vertical aggregation in a temporary table F_V and then horizontal aggregations are indirectly computed from F_V. We now present the direct aggregation method. Horizontal aggregation queries can be evaluated by directly aggregating from F and transposing rows at the same time to produce F_H. First, we need to get the unique combinations of $R_1, \ldots, R_k$ that define the matching boolean expression for result columns. The SQL code to compute horizontal aggregations directly from F is as follows. Observe $V()$ is a standard (vertical) SQL aggregation that has a ”case” statement as argument. Horizontal aggregations need to set the result to null when there are no qualifying rows for the specific horizontal group to be consistent with the SPJ method and also with the extended relational model. This statement computes aggregations in only one scan on F. The main difficulty is that there must be a feedback process to produce the”case” Boolean expressions.

C. Pivot Method

We consider the PIVOT operator which is a built-in operator in a commercial DBMS. Since this operator can perform transposition it can help evaluating horizontal aggregations. The PIVOT method internally needs to determine how many columns are needed to store the transposed table and it can be combined with the GROUP BY clause. The basic syntax to exploit the PIVOT operator to compute a horizontal aggregation assuming one BY column for the right key columns (i.e. k = 1). Notice that in the optimized query the nested query trims F from columns that are not later needed. That is, the nested query projects only those columns that will participate in F_H. Also, the queries can be computed from F_V.

D. Enhance SPJ, Case and Pivot Method

The enhancement proposes to apply masking on SPJ, Case and Pivot methods to secure the data and provide privacy to the data being aggregated. Any type of masking or encryption can be used to protect the data being published. We show that providing binary encryption i.e converting data into binary masked value will increase the memory space required to store the data in binary form. The better method would be to convert it back into hex decimal format to reduce the memory space and we show the comparative results of our evaluation.

IV. NUMERICAL RESULTS

The concept of this paper is implemented and different results are shown below, The proposed paper is implemented in Java technology on a Pentium-IV PC with minimum 20 GB hard-disk and 1GB RAM. The propose paper’s concepts shows efficient results and has been efficiently tested on different Datasets.

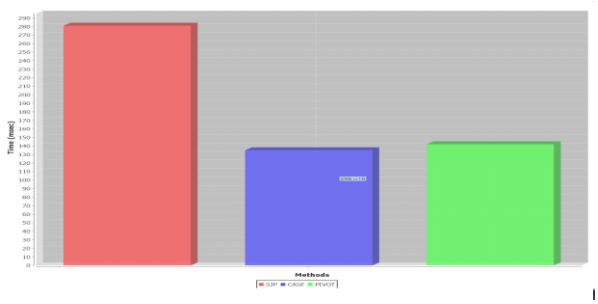

Fig. 3 Time Consumed in Three Methods.

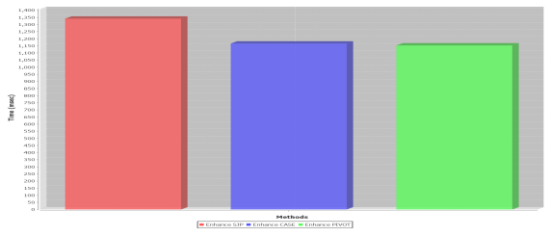

Fig. 4 Time Consumed in Enhanced Methods

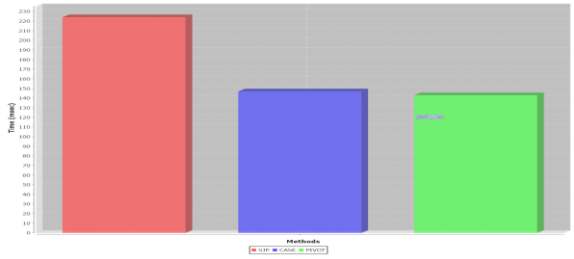

Fig. 5 Time consumed in Three Methods for different aggregate function

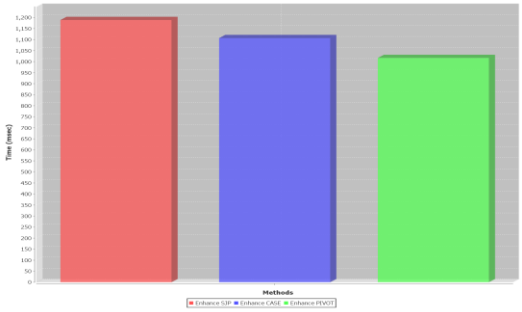

Fig. 6 Time consumed in Enhanced Method for different Aggregate function

V. CONCLUSION

We introduced a new class of extended aggregate functions, called horizontal aggregations which help preparing data sets for data mining and OLAP cube exploration. Specifically, horizontal aggregations are useful to create data sets with a horizontal layout, as commonly required by data mining algorithms and OLAP cross-tabulation. Basically, a horizontal aggregation returns a set of numbers instead of a single number for each group, resembling a multi-dimensional vector. We proposed an abstract, but minimal, extension to SQL standard aggregate functions to compute horizontal aggregations which just requires specifying subgrouping columns inside the aggregation function call. From a query optimization perspective, we proposed three query evaluation methods. The first one (SPJ) relies on standard relational operators. The second one (CASE) relies on the SQL CASE construct. The third (PIVOT) uses a built-in operator in a commercial DBMS that is not widely available. The SPJ method is important from a theoretical point of view because it is based on select, project and join (SPJ) queries. The CASE method is our most important contribution. It is in general the most efficient evaluation method and it has wide applicability since it can be programmed combining GROUP-BY and CASE statements. We proved the three methods produce the same result. We have explained it is not possible to evaluate horizontal aggregations using standard SQL without either joins or ”case” constructs using standard SQL operators. Our proposed horizontal aggregations can be used as a database method to automatically generate efficient SQL queries with three sets of parameters: grouping columns, subgrouping columns and aggregated column.

REFERENCES

[1] C. Cunningham, G. Graefe, and C.A. Galindo-Legaria. PIVOT andUNPIVOT: Optimization and execution strategies in an RDBMS. In*Proc. VLDB Conference*, pages 998–1009, 2004.

[2] G. Luo, J.F. Naughton, C.J. Ellmann, and M. Watzke. Locking protocols for materialized aggregate join views. *IEEE Transactions on Knowledgeand Data Engineering (TKDE)*, 17(6):796–807, 2005.

[3] C. Ordonez. Vertical and horizontal percentage aggregations. In *Proc.ACM SIGMOD Conference*, pages 866–871, 2004.

[4] C. Ordonez. Data set preprocessing and transformation in a databasesystem.*Intelligent Data Analysis (IDA)*, 15(4), 2011.

[5] C. Ordonez. Integrating K-means clustering with a relational DBMSusing SQL.*IEEE Transactions on Knowledge and Data Engineering(TKDE)*, 18(2):188–201, 2006.

[6] C. Ordonez and S. Pitchaimalai.Bayesian classifiers programmed inSQL.*IEEE Transactions on Knowledge and Data Engineering (TKDE)*,22(1):139–144, 2010.

[7] C. Ordonez. Statistical model computation with UDFs.*IEEE Transactionson Knowledge and Data Engineering (TKDE)*, 22, 2010.

[8] H. Wang, C. Zaniolo, and C.R. Luo.ATLaS: A small but complete SQLextension for data mining and data streams. In *Proc. VLDB Conference*,pages 1113–1116, 2003.

[9] C. Ordonez. Horizontal aggregations for building tabular data sets. In*Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*,pages 35–42, 2004.